ECE 575 Final Project: Implementing a 5-stage processor pipeline in Python

Robert Werthman
ECE 575 Section 02
University of Michigan Dearborn

***Abstract*—This document describes an implementation of a 5-stage instruction processing pipeline in the programming language python. The processor, pipeline stages, instruction, and data memory can be represented in Python using object-oriented programming and python structures like lists and dictionaries.**

I. Introduction

There are multiple ways to execute instructions in a processor. One way is to execute one instruction in one clock cycle. This is known as single cycle execution. In single cycle execution the processor datapath which is defined as all the elements necessary to implement a set of instructions is designed such that all instructions that are part of the instruction set architecture are executed in a single cycle [2, p. 1]. An often used analogy to the execution of an instruction is doing laundry as mentioned in [1, p. 272]. Let's say the instruction you wanted to execute was to wash dirty cloths and put them away. In a single cycle datapath the steps would be

1) Put dirty cloths in washer.
2) Once washer is done put dirty cloths in dryer.
3) When dryer is done put clothes away

As you can see each step depends on the completion of the previous step. Imagine you had a roommate who also folded his/her clothes before putting them away was coming after you to do laundry. The roommate has an additional step in their datapath and would have to wait for everything to be done before starting their load of laundry. If you and your roommate were driven by a single clock cycle the cycle would have to last as long as the time to complete your roommate's clothes washing steps. This is because in a single cycle datapath the cycle must last as long as the instruction that has the longest path to complete [1, p. 271]. It was determined this was not very efficient and a more efficient way of executing instructions was developed. This better way to execute instructions in a processor called the pipelined datapath.

Think of an instruction as a series of steps. All steps need to be completed in order for the instruction execution to be completed. In a pipelined datapath one step of an instruction is executed every clock cycle instead of all of the steps of an instruction being executed every clock cycle. Using the clothes washing analogy again, the steps in a pipelined datapath would be

1) Put dirty cloths in washer.
2) Once washer is done put dirty cloths in dryer. Roommate puts dirty clothes in washer.
3) Once dryer is done put clothes away. Roommate puts clothes from washer into dryer.
4) Roommate fold clothes.
5) Roommate puts clothes away.

You may have noticed the roommate does not have to wait for you clothes to be completely done. The roommate can use the washer as long as you don't have any clothes in it. This was found to be a more efficient datapath design for instructions than the single cycle datapath. As noted in [1, pp. 286] the steps of an instruction in a pipelined datapath are made into 5-stages

1) IF: Instruction fetch
2) ID: Instruction decode and register file read
3) EX: Execution or address calculation
4) MEM: Data memory access
5) WB: Write Back

This stages can be modelled in Python as objects connected to each other. The rest of this document describes the implementation of this modelling.

II. Architecture

The Python implementation is made of the 5 pipeline stages mentioned in the introduction. These are represented as Python classes. Each of these stages consist of additional components.

- instruction memory
- data memory
- a register file
- an ALU (arithmetic logic unit)
- a PC (program counter)
- a control unit

All of these components belong to different stages of the pipeline except for the control unit which is used in almost all stages of the pipeline. A class diagram of the architecture can be seen in figure 1.

A. IF: Instruction Fetch

The instruction fetch part of the pipeline takes the program counter (PC) and fetches the instruction at the address of the program counter [1, pp. 300]. It increments the program counter to the next address. These things are done in a class method called execute which is pipeline stage contains

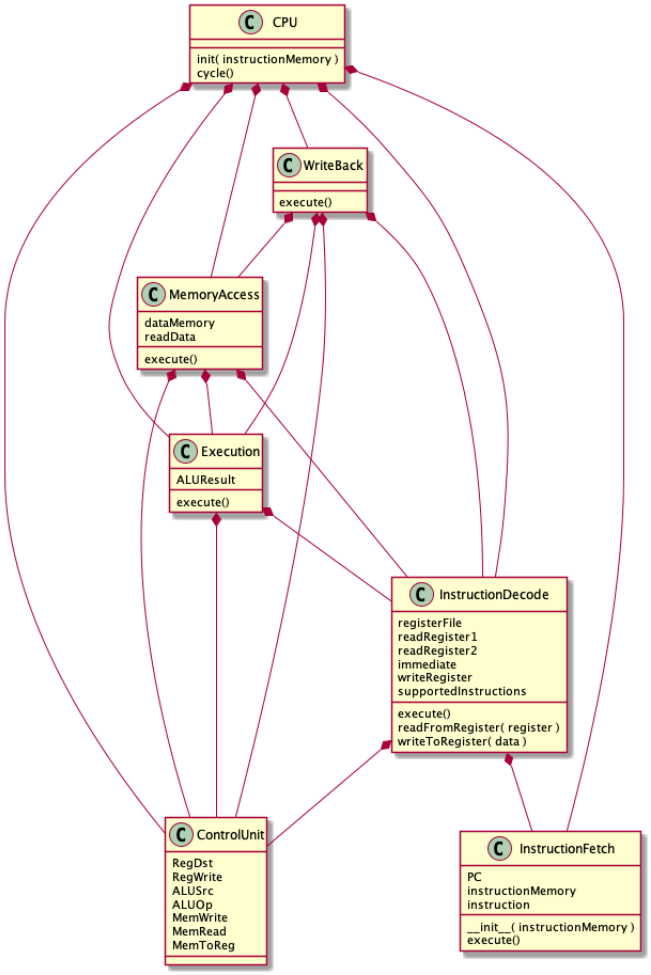

Fig. 1. Class Diagram of Python Implementation

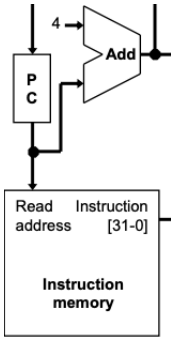

Fig. 2. Instruction Fetch Stage [2]

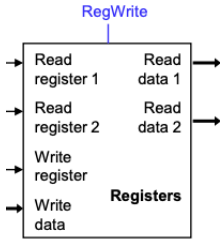

Fig. 3. Instruction Decode Stage [2]

1) Instruction Memory: Instruction memory is represented as a Python list that is passed into the instruction fetch class at initialization. Each instruction itself is also a Python list. Depending on the instruction type the list is made up of the opcode, source and destination registers, address/immediate, etc. An example of an add instruction in python

```
# An add instruction
instruction = [ "add", "r1", "r2", "r3" ]
```

which means [3, pp. 194]

op	rs	rt	rd

To access an instruction memory you use an index in the list. For example

```
# Get the first instruction from instruction memory
instruction = instructionMemory[0]
```

2) Program Counter: The program counter is a python variable that is initialized as 0 and is increment by 1 for every call to execute method of the instruction fetch class.

B. ID: Instruction Decode

In the instruction decode stage of the pipeline the control signals are set in the control unit based on the opcode in the instruction. The instruction is referenced from the instruction fetch class through a member variable. The read registers, rs and rt, are stored in the class as well as the write register, and the immediate/address value if one exists. These come from the instruction from the instruction fetch stage. Only add, load word (lw), and store word (sw) are supported instructions at this time. The supported instructions are mapped to their instruction type in a dictionary.

```
self.supportedInstrutions = {
            "add" : "rtype",
            "lw"  : "itype",
            "sw"  : "itype"
        }
```

The instruction type is used to determine if there will be an immediate/address in the instruction or not. There are 2 methods that are a part of this stage. They are the readFromRegister and writeToRegister methods simliar to the read data and write data in [1, pp. 301]. The instruction decode stage has a register file.

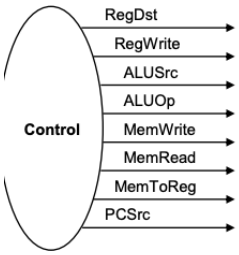

Fig. 4. Datapath Control Unit

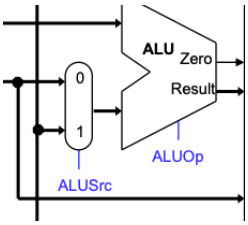

Fig. 5. Execution Stage

1) Register File: The register file is a dictionary with the register has the key and value of the register as the value.

```
self.registerFile = {
            "r1" : 2,
            "r2" : 1,
            "r3" : 0
        }
```

Only 3 registers are supported at this time.

2) Control Unit: The control unit is a class that contains just member variables which are the control signals to the pipeline. The signals are set to their values depending on what the opcode of the instruction is. The control unit is seen in figure 4

C. EX: Execution

In the execution state of the pipeline we compute the result of arithmetic between 2 registers, or compute an address to load or store a value in data memory. The execution stage depends on control signal ALUOp to tell it was arithmetic operation to perform. It also depends on ALUSrc to tell it what should be the source of the second operand. The execution statge is figure 5 Right now only add is supported but this still allows for an add r-type instruction or the load word or store word i-type instructions.

The result of the arithmetic operation is stored as a class instance member variable that both the Memory Access and Write Back stages will use.

D. MEM: Memory Access

In the memory access stage you can read or write data to data memory or not do anything if the instruction is an r-type instruction. This stage is comprised of the data memory and the ability to used an address to read and write data to.

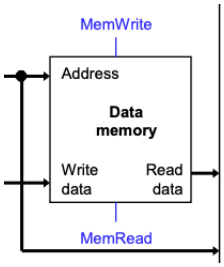

Fig. 6. Memory Access Stage [2]

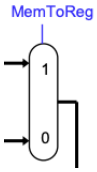

Fig. 7. Write Back Stage [2]

1) Data Memory: Much like the instruction memory, data memory is represented as a list of integers.

```
# Data memory
dataMemory = [1, 2, 3, 4]
```

The ALUResult that comes from the Execution stage is used as an index into the data memory list. Depending on the values of the control signals MemRead or MemWrite you will be either reading or writing to the data memory. An if/else statement is used with control signals as the conditions to read or write to data memory.

E. WB: Write Back

The Write Back is the simplest of all of the stage but it is the stage that depends on the most previous stages. As depicted in [1, pp. 301] this stage has a MUX that is controlled by the MemToReg control signal. The value of this signal depends on the instruction being executed. The MUX is represented by an if/else statement with the MemToReg as the if condition.

```
if controlUnit.MemToReg:
    ID.writeToRegister( MEM.readData )
else:
    ID.writeToRegister( EX.ALUResult )
```

F. CPU

The CPU is the main class that creates and contains all of the pipeline stages. It is the class that contains the initialization of the instruction memory, passes it to the instruction fetch stage, and starts the execution of the pipeline.

The the execution of the stages is done in a method called *cycle*. For each instruction in instruction memory execute the pipeline stages in order.

```
for instruction in instructionMemory:
    IF.execute()
    ID.execute()
    EX.execute()
    MEM.execute()
    WB.execute()
```

III. Testing and Verification

Test driven development was used when developing this project. This means tests were written, the tests failed, and then code was written to make the tests pass. For each class a python file containing containing tests was created. These tests were used to verify the behavior of each of the methods of the class. The main integration driver were the tests for the CPU. In those tests I would create a CPU that contained all of the stages, create an instruction memory with one instruction, and have the CPU process the instruction through all of the stages. At the end I would check to see if a register has the expected value or if the data memory was written in the right location with the expected value

IV. Open Issues And Future Work

References

[1] J. Hennessy and D. Patterson, *Computer Organization And Design: The Hardware/Software Interface*, 5th ed. Waltham, MA: Elsevier, 2014.

[2] L. Ceze, "A single-cycle MIPS processor". University of Washington, Computer Science and Engineering, CSE 378, Winter 2009. [Online]. Available: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec07.pdf. [Accessed December 03, 2019]

[3] P. Gillard, "The MIPS instruction set architecture". Memorial University, Department of Computer Science, CS 3725. [Online]. Available: http://web.cs.mun.ca/~paul/cs3725/material/review.pdf. [Accessed December 03, 2019]